COMPARISON OF DISTRIBUTED TRAINING ARCHITECTURE FOR CONVOLUTIONAL NEURAL NETWORK IN CLOUD

BY

DONGWEI SHI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Wen-mei Hwu

ABSTRACT

The rapid growth of data and ever increasing model complexity of deep neural networks (DNNs) have enabled breakthroughs in various artificial intelligence fields such as computer vision, natural language processing and data mining. The training process of the DNN is a computationally intensive application that can be accelerated by parallel computing devices such as graphic processing units (GPUs) and field programmable gate arrays (FPGAs). However, sometimes the amount of the training data or the size of the model might exceed what can be efficiently trained or loaded by a single machine. Distributed deep learning training addresses this issue by spreading the computations over several machines. Due to the internode communication and other overheads in distributed computing infrastructure, the performance improvements are not directly proportional to the number of machines. This thesis will study the computation time, memory, bandwidth, and other resources that are required to perform distributed deep learning. The approach of this work is to implement and deploy several data parallelism distributed deep learning algorithms on Google Cloud Platform (GCP) and then to analyze the performance and compare the communication overhead between different algorithms. The results obtained in this research yield the Ring All-Reduce architecture, a bandwidth-optimal communication operation used for distributed deep learning, which outperforms the Parameter Server architecture, a many-to-one architecture, on scalability. In addition, system usage information reported from GCP is leveraged to identify the bottleneck of a neural network training on distributed architecture.

To my parents, for their endless love and unconditional support.

ACKNOWLEDGMENTS

I would like to thank my thesis research adviser, Professor Wen-Mei W. Hwu, for his inspiration and support. His devotion and hard work have always been examples to follow. I appreciate the opportunity to work with him as my thesis adviser and take his courses on applied parallel programming and many-core algorithms which inspired this thesis.

I would also like to thank the NCSA Gravitational Research Group members, past and present, for their tremendous help and camaraderie.

I want to thank the University of Illinois Graduate College for providing me the opportunity to study and complete my master's degree here.

Finally, I would like to thank my parents for their endless love and unconditional support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The emergence of deep learning (DL) has significantly extended the applications of real-world artificial intelligence tasks such as computer vision, natural language processing and data mining. Unlike traditional machine learning algorithms that require feature engineering to interpret the data domain and make patterns meaningful to the algorithm itself, a DL algorithm tries to learn high-level features from data in an incremental manner. This approach eliminates the need for feature engineering for data interpretation, which is extremely difficult.

The recent success of DL is largely attributed to the exponential growth of data and continuously increasing computing power. However, in some cases, the training data and the number of parameters in the network are oversized to a single machine. Distributed deep learning addresses this problem. By leveraging a distributed system, we can not only train deeper neural networks, but also accelerate our training process.

Applying a distributed system to deep learning training is an active research area in recent years. There are two mainstream parallel computing strategies: model parallelism and data parallelism. According to ideal Amdahl's law, the performance speed-up of the training process is linear to the number of machines. However, in most practical cases, the distributed deep learning algorithm is not merely a multiple of the number of the machines. A major challenge comes from the communication overheads in the distributed system. Particularly, both the iterative optimizing process (back-propagation) of the DL algorithm and the tremendous size of the training data require a huge amount of communication between computing nodes in a distributed system.

In this research, we will empirically analyze the performance and the scalability of mainstream distributed DL architectures in Google Cloud Platform, the same distributed computing infrastructure that Google uses.

The rest of the thesis is organized as follows: Chapter 2 presents some necessary background and related work, Chapter 3 describes the experimental setup and implementation, Chapter 4 reports the results and statistics of different experiments. Finally, Chapter 5 concludes the thesis.

CHAPTER 2

BACKGROUND

2.1 Neural Networks

The original motivation of artificial neural networks (ANNs) was to model the biological structure of the human brain. However, this biological approach has achieved better results in some machine learning tasks than statistical machine learning models. The basic computational module of an ANN is the perceptron. The mathematical operation of a perceptron could be simply written as $w_0 x_0$, where x_0 represents the signals that travel along the axons and w_0 stands for the dendrites of the other neurons based on the synaptic strength at that synapse. In machine learning problems, x_0 is the input feature and w_0 is the learnable weights. As shown in Figure 2.1, with multi-dimension data, the matrix notation of a perceptron is $\sum_1^N x_i w_i$. To make this linear predictor nonlinear, an activation (non-linear) function f is introduced. By connecting layers of perceptrons in a feed-forward manner, the ANN is able to approximate any arbitrary function.

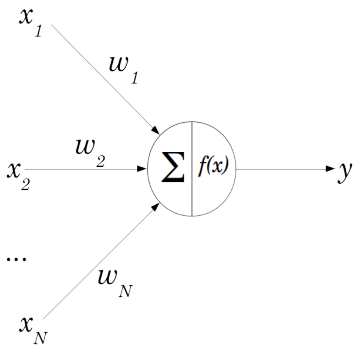

Figure 2.1: A perceptron

2.2 Convolutional Neural Networks

A convolutional neural network (CNN) is similar to the NN architecture introduced in section 2.1. As shown in Figure 2.2, the NN receives a single vector of input and forwards it to a series of hidden layers. Perceptrons in each layer of an NN are fully collected. In other words, each connection represents a learnable weight to the NN algorithm. For example, if the dimensions of an input RGB image are $200 \times 200 \times 3$, the learnable weights would be 120000. Thus, the NN architecture does not scale well to images or high-dimensional data applications. Unlike NN architecture, the CNN architecture is designed to address this issue.

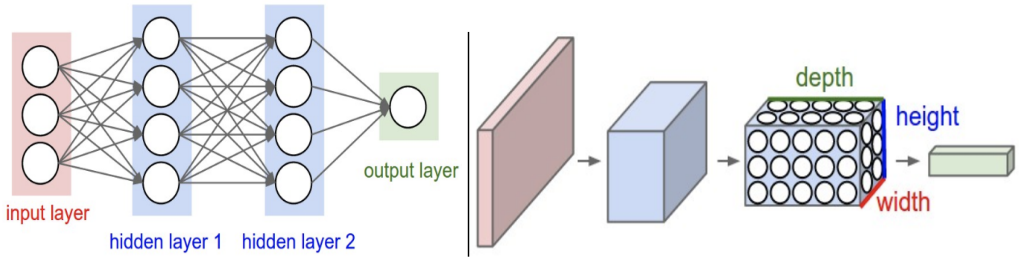

Figure 2.2: Neural network against ConvNet [1]

Most CNN models consist of a series of layers that distill important features of the input image in an incremental manner. A few basic types [1] of layers could be used to construct a CNN model.

- Convolutional Layer: The convolutional layer is the core building block that requires most computational resource. The objective of the convolutional operation is to extract the high-level features such as shapes or contours. Recent studies [2, 3] have shown that the few first convolutional layers are responsible for learning low-level features such as colors and edges of the input image. With increasing the convolutional layers, the CNN architecture is expected to adapt high-level features.

- Pooling Layer: Similar to the convolutional layer, the objective of the pooling layer is to reduce the spatial size of a captured feature. It is common to insert a pooling layer between convolutional layers.

- ReLU Layer: Rectified linear unit is an element-wise activation func-

tion. The ReLU layer increases the non-linearity of the CNN architecture.

The CNN architecture combined with different layers can approximate on arbitrary mathematical model for computer vision tasks.

2.3 Mini-batch Stochastic Gradient Descent

In machine learning and DL training processes, gradient descent (GD) is the most common optimization method [4]. By giving a constant learning rate, the model computes the current gradient and updates the learnable weights. Since the gradient computation is performed across the whole dataset, GD optimizer always gives accurate gradient estimation and robust weight updates. However, there are two disadvantages when applying GD optimizer. First, given a large dataset, the gradient computation is time-consuming. Second, the loss function of the DL network is not a convex optimization problem. Based on convex optimization theory [5], the GD optimizer guarantees convergence only on local optimum, a not global optimum. Thus, using GD optimizer to train a DL neural network, the model might eventually converge to a local optimum which is not desired.

Stochastic gradient descent (SGD) optimizer computes each gradient by using one data sample. This optimizer not only improves the computation time but also is well suited to online training. However, the gradient computed from a single data sample is inaccurate. Thus, the learning rate needs to be very small. Since most modern computing devices such as CPUs and GPUs are multi-threaded architectures, a single data sample finds it hard to utilize the CPU and GPU during the training process. Thus, SGD optimizer leads to a waste of computational resources.

The method to address the disadvantages of GD and SGD is mini-batch SGD [6]. In each training epoch, the mini-batch SGD tends to use multiple samples (mini-batch) to compute the gradient. In this way, the estimation of the gradient is more accurate than calculating the gradient against only one training sample (SGD). At the same time, a batch of the training data can utilize the CPU and GPU computational resources efficiently. Furthermore, since mini-batch SGD reduces gradient noise, it suppresses the disadvantage of GD optimizer that sticks to local optimum. The algorithm is as follows:

Algorithm 1 Mini-batch Stochastic Gradient Descent

```
1: procedure MINI-BATCH SGD
2:     Randomly initialize learn-able weights w
3:     Set learning rate η
4:     while the approximate optimum not converge do
5:         Randomly shuffle examples in the training set.
6:         Divide data into batches
7:         for each batch over batches do
8:             Select directions d_k on batch
9:             w ← w + ηd_k
10:        end for
11:    end while
12: end procedure
```

In Algorithm 1, descent direction d_k satisfies $\nabla f(w)^T d_k < 0$, where $f(w)$ is the function that the DL model is trying to approximate.

2.4 Model Parallelism

As one of the distributed training strategies, model parallelism is an approach to deploy and implement the deep learning model on multiple machines. Deep learning models can have billions of parameters that cannot fit into a single computing device's memory. Model parallelism addresses this issue by splitting the model on multiple computing devices. Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning [3]. As shown in Figure 2.3, in order to train a deep LSTM model in a parallel manner, it is necessary to spread the different parts of the model to several machines. As discussed above, deep learning consists of multiple layers. Leveraging model parallelism, we deploy different layers on different machines. However, the computation between layers has constraints: forward requires the current layer to wait for the previous layer's output, and back-propagation bases the computational results of the current layer on those of the subsequent layers. Hence, unless the model is so large that it cannot fit into the system memory, model parallelism is seldom applied during the distributed deep learning training process due to the serial communication constraints between layers. However, if the model itself contains a parallel execution module, for example the inception mod-

ule in GoogLeNet [7], there is no doubt that applying model parallelism can accelerate the training process.

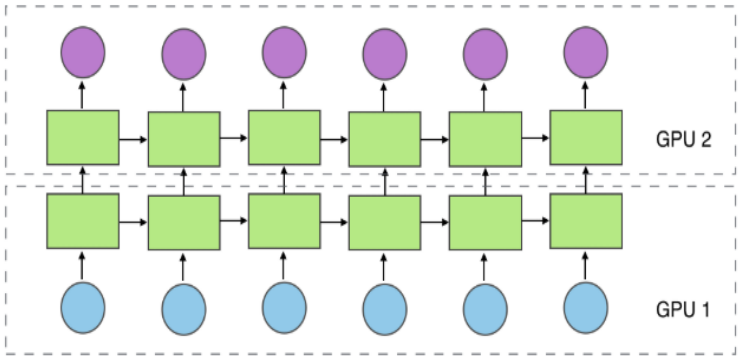

Figure 2.3: Each LSTM layer is assigned to a different GPU. After GPU 1 finishes computation, it passes its output to GPU 2 for further processing. [8]

2.5 Data Parallelism

Data parallelism [9] deploys a model replica on each computing device and computes the gradient using a different batch of the training data. As shown in Figure 2.4, each device contains a completed model replica that can be trained individually. Compared with the model parallelism, data parallelism is able to support larger training and provide better scalability. Hence, data parallelism is a prevailing strategy adopted by many distributed deep learning applications.

In section 2.4, we learned that the training process of a deep learning model is iterative. In each training epoch, the forward algorithm computes the prediction value across the training data based on current hyperparameters. Then the back-propagation updates the learnable weights by computing the gradient of the loss function. In distributed deep learning, different computing devices such as CPUs and GPUs are able to execute this iterative process on different mini-batches of the training data. The major difference is in the way of updating the learnable weights.

Data parallelism can be either synchronous or asynchronous. Synchronous data parallelism means every device applies identical hyperparameters for model training process. After completing the gradient computation process

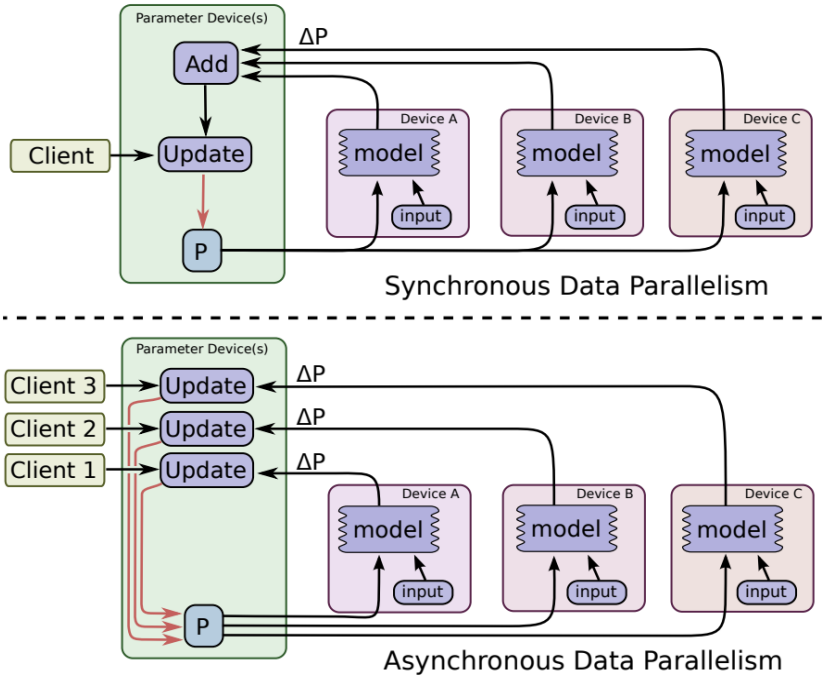

Figure 2.4: Synchronous and asynchronous data parallel training [10]

(training) individually on each computing device, a parameter device(s) is responsible for gathering the weights computed by each computing device and using the average of the collected weights to update the model's learnable parameters in each iteration. It is similar to combining mini-batches on all the devices into a large batch and training the deep learning model on this large batch of the data. Although synchronous data parallelism appears to be effective on large-scale deep learning training processes, the computational capacity and the communication overheads across each node in a cluster need to be balanced. Since it requires waiting for the last device to finish the computation, synchronous data parallelism is slower than asynchronous data parallelism. Unlike synchronous data parallelism, asynchronous data parallelism directly updates the learnable weights without waiting for other devices to complete the training process on individual mini-batches of data. As shown in Figure 2.4, in each iteration, different devices will only load the latest updated parameters. Since the parameters are updated asynchronously, there is a high probability that each device is getting different values of the parameters.

Overall, asynchronous data parallelism is faster. However, a major challenge comes from stale gradients. At the beginning, each device sets off the training process using identical parameters. Because of updating weights asynchronously, a device might find out that the parameters of the model have been updated by another device. In this situation, the gradients computed by that device are obsolete. In other words, the device computes stale gradients. Because of the stale gradients, the model might find a sub-optimal solution (local optimum), which should be avoided in training a deep neural network.

2.6 Parameter Server

Parameter Server [11] (PS) is one of the most commonly used data parallelism architectures in distributed deep learning training. There are two types of node in the cluster of PS architecture: Parameter Server and Worker. Parameter Server is responsible for storing the parameters of the deep learning model. Worker node is responsible for calculating the gradient for the loss function of the model. In each interaction, Worker node gets the learnable weights from Parameter Server. Worker node computes the gradient based on the learnable weights gotten previously. Parameter Server gathers the newly computed gradient from Worker nodes and updates the learnable weights according to these gradients. Then, Parameters Server broadcasts the updated learnable weights for the next iteration in the training process. In Figure

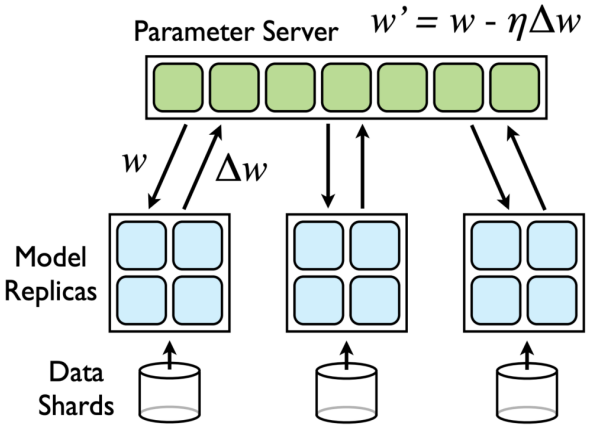

Figure 2.5: Parameter Server architecture [10]

2.5, there are three Worker nodes and one Parameter Server. Each Worker node contains a replica of the model and a Data Shard, a mini-batch of the training data. First, each Worker node gets w, learnable weights, from the Parameter Server. After the calculation of Δw, each Worker node sends Δw back to the Parameter Server. Then, Parameter Server computes the new learnable weights by $w' = w - \eta\Delta w$ for the next iteration of the training.

2.7 Ring All-Reduce

In PS architecture, as Worker nodes increase, the communication overheads between Worker nodes and Parameter Server nodes will be the bottleneck of the distributed training system. The Ring All-Reduce architecture [12] is designed to address this issue. In Ring All-Reduce architecture, the centralized node (Parameter Server) that gathers information from Worker nodes is eliminated. Each node in the Ring All-Reduce architecture is a Worker node. Computing devices in Ring All-Reduce architecture form a topological ring. Each device accepts the parameter information from its predecessor and sends the latest computed parameters to the successor in the ring. Thus, this architecture takes advantage of communication bandwidth between devices. Figure 2.6 demonstrates this communication pattern. The algorithm

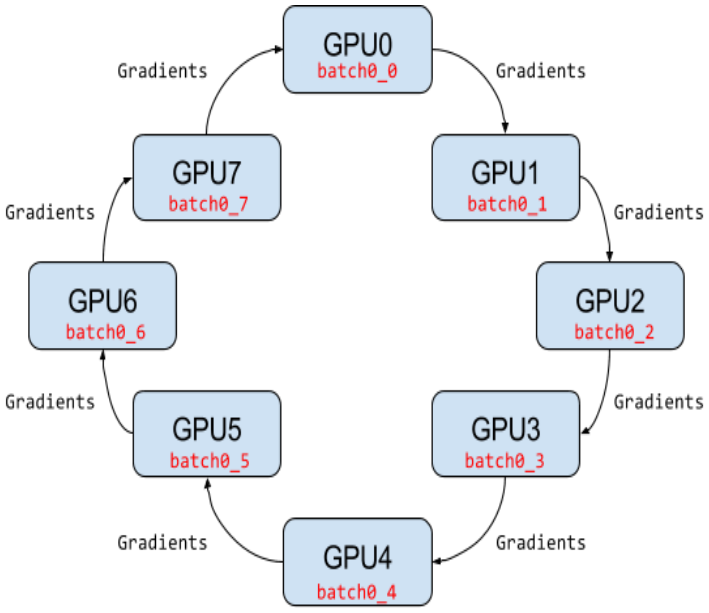

Figure 2.6: Ring All-Reduce Architecture

of using Ring All-Reduce architecture to perform gradient updates is shown as follows:

- Split the parameters or learnable weights on each Worker node into data chunks.
- In Scatter-Reduce phase, after $Number of Devices - 1$ iterations of aggregation with its local copy and data transfer between neighbors, each Worker node computes gradient once according to the aggregated chunks from all Worker nodes in the ring.
- In All-Gather phase, after $Number of Devices - 1$ iterations of aggregation with its local copy and data transfer between neighbors, each Worker node receives and sends the gradient computed in previous step from its predecessor and successor.
- Merge the chunks in each Worker node and compute the sum of the total gradient. Then, divide the total gradient by $Number of Devices$. In this way, we obtain the average gradient used for updates in one training epoch.

An example of the above process is presented in Figure 2.7. Each Worker with 3 data buffers communicates with two of its peers 4 times. During this communication, a node sends and receives chunks of the data buffer. In the first 2 iterations, received values are added to the values in the nodes buffer. In the second 2 iterations, received values replace the values held in the nodes buffer. The communication bandwidth between Worker nodes in the topological ring does not increase with the number of nodes in a distributed system. The total amount of data that transfers once among the Ring All-Reduce architecture is:

$$2 \cdot \frac{Number of Devices - 1}{Number of Devices} \cdot Size of chunk \approx 2 \cdot Size of chunk$$

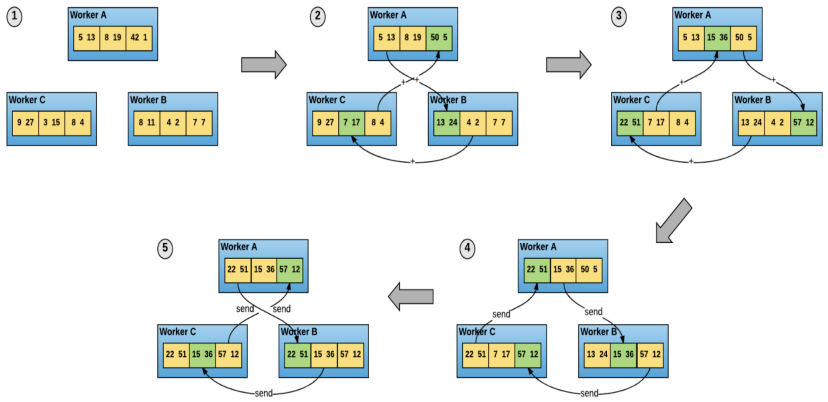

Figure 2.7: An example to illustrate the gradient updates with three Worker nodes

Compared to the PS architecture, the Ring All-Reduce architecture is bandwidth-optimal, since each Worker node maximizes the communication bandwidth between neighbors. Furthermore, due to the nature of back-propagation, the optimizer first computes the gradients from deeper layer to shallower layer in a deep neural network. The Ring All-Reduce algorithm can take advantage of this feature and further accelerate the training process on a large scale.

2.8 TensorFlow

Tensorflow [10] is an open-source framework for machine learning and deep learning applications. Tensorflow has support for both CPUs and GPUs on one node or a cluster with multiple nodes. The main entities of the Tensorflow programming framework are:

- Graph: Tensorflow computation process can be described as data flow graphs, where graph edges represent data flow and vertices represent computational operations.
- Operation: Each operation object represents a vertex in the graph. Each vertex is responsible for performing addition, multiplication, or some more complex operations. Each vertex takes a tensor as input and produces a tensor as output.

- Tensor: Tensors are edges of a data flow graph. Tensors hold feature maps or model's weights which are calculated in a session. A tensor is a multidimensional vector with a rank.

- Session: This is an entity that holds the information and setups about the TensorFlow graph. Sessions perform computations described by the data flow graph.

2.9 Distributed TensorFlow

TensorFlow supports both data and model parallelism and allows both synchronous training and asynchronous training. The code on a single node is consistent with the code on a cluster of nodes. Thus, the developers do not need take into consideration the infrastructure of TensorFlow. It is convenient for developers to maintain and scale the code to a cluster. In distributed TensorFlow, the computational nodes perform training process and computations are described as a cluster. Each cluster contains multiple servers. Each server in a cluster performs a task. Server and task are in one-to-one correspondence. Hence, a cluster of servers is equivalent to a cluster of tasks. TensorFlow programming framework merges similar tasks into a job. For instance, in Parameter Server architecture, we recognize the tasks that perform parameter gathering as Parameter Server and the tasks that perform gradient computation as Worker nodes. Thus, from TensorFlow aspect, a cluster is a set of jobs as well. However, the actual functionalities depend on the implementation of servers.

In TensorFlow, a hash table is applied to index the job type on each hosting machine. In a specification hash table, the hosting machine, either Worker or Parameter Server, is a string including the information of network addresses in a cluster.

2.10 Related Work

Both big data and deep learning make the training process time-consuming; thus it is an active research field in recent years. The implementation of

distributed deep learning model not only accelerates the training process but also provides an efficient way to find optimal hyper-parameters for the model.

Using distributed computing to accelerate deep learning training has enjoyed a surge of interest in industry. In 2017, Facebook announced that using 32 servers with 256 GPUs on board, they are able to train Resnet-50 on the ImageNet dataset in one hour, a task that previously took two weeks. They used large mini-batches of the data, 8192 images per batch, to train the model. The study [13] shows that the learning rate is proportional to the size of mini-batch. Facebook applied data parallelism and PS architecture in this work. On the other hand, Baidu used data parallelism strategy and Ring All-Reduce architecture to perform distributed training. They used 40 Worker nodes to form a topological ring. The findings show the acceleration of training is proportional to the increment of nodes. Baidu submitted their Ring All-Reduce approach to the TensorFlow community. Uber developed Horovod [14], an open-source Ring AllReduce distributed deep learning library, for TensorFlow.

Before the invention of TensorFlow, Google applied DistBelif [10] as a major distributed deep learning training framework which uses Parameter Server architecture. TensorFlow derives from DistBelif, and thus TensorFlow originally supports the Parameter Server architecture. As one of the most popular and active deep learning frameworks, TensorFlow supports both CPUs and GPUs on one node or a cluster with multiple nodes from version 0.8. TensorFlow officially supports Ring All-Reduce architecture in version 1.11.

Technology companies use various distributed infrastructures to implement their own distributed deep learning algorithms. This study aims to horizontally analyze different distributed deep learning approaches on the same infrastructure. Thanks to increasingly convenient access to public cloud, such as Amazon AWS, Google Cloud, and Microsoft Azure, deploying various distributed deep learning strategies and architectures on the same distributed infrastructure becomes feasible to researchers.

This study uses an 8-node cluster deployed on Google Cloud. Each node contains an NVIDIA Tesla K80 GPU [15]. Tests on multiple convolutional neural network models were performed with different distributed learning strategies and architectures.

CHAPTER 3

IMPLEMENTATION

The objective of using a distributed deep learning algorithm is to accelerate the deep learning training process. An accelerated training process helps find the optimal hyperparameters, which are keys to the performance of a deep learning model. To implement an efficient distributed deep learning algorithm, it is necessary to consider the communication overheads between computing nodes in a cluster. Two widely used distributed deep learning algorithms, Parameter Server and Ring All-Reduce, are implemented and deployed on an identical cluster setup equipped with 8 GPUs on Google Cloud Platform.

3.1 Models

This study benchmarks multiple convolutional neural networks (CNNs). AlexNet [16], VGG16 [17], VGG19 [17], ResNet50 [18], and ResNet152 [18] are used to perform image classification task on CIFAR-10 datasets with different distributed training strategies and architectures. Analysis and comparison are performed on the run-time with different CNN models. Size and performance of selected CNN models are presented in Table 3.1.

Table 3.1: Models used in this experiment with their model size, accuracy and GFLOPs

Models	Model size (MB)	Top-1(%)	Top-5(%)	GFLOPs
AlexNet	233	57.0	80.3	0.7
VGG16	528	70.5	90.0	15.5
VGG19	548	71.3	92.7	19.6
ResNet50	98	75.8	92.9	3.9
ResNet152	230	77.6	93.8	11.3

3.2 Datasets

CIFAR-10 is one of the classical datasets for image classification. This dataset contains $60,000$ images with 32×32 RGB images. Among the $60,000$ images, $50,000$ are training dataset and $10,000$ are testing dataset. CIFAR-10 dataset has 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each class has 6000 images. Furthermore, each image in a class only contains one object. For instance, an airplane class image contains only one airplane and an automobile class image contains only one automobile. It is impossible for an image to contain both airplane and automobile objects in the CIFAR-10 dataset.

This dataset aims to validate the correctness of the distributed deep learning algorithms, instead of the robustness of the CNN models. The CIFAR-10 dataset is integrated with TensorFlow, which is easy to manipulate and import for use.

3.3 Cluster Information

The GCP provides users flexibility to define a cluster with different numbers of nodes. This experiment applies an 8-node cluster equipped with CPUs and GPUs. The hardware specifications for each node are:

- CPU: an Intel Xeon E5 [19] virtual CPU
- GPU: an NVIDIA K80 [15] GPU
- Memory: 3.75 GB
- Storage: 10GB disk storage

Each node runs on Ubuntu 16.04 lts OS image. Each node has the following software and programming frameworks installed for distributed deep learning algorithm:

- Python 2.7
- CUDA toolkit 10.0
- cuDNN 7.4.1.5
- TensorFlow 1.12

3.4 Implementation of Experiment

Distributed TensorFlow treats a series of tasks as a job [10]. In TensorFlow, we use name, a string, to identify the job and index, an integer, to describe a task. By concatenating name with an index, each task in a cluster has a unique identification label. In a distributed system, each task executes on a different node or computing device in a cluster. Fortunately, TensorFlow provides an API for the user to give the description of a cluster by using tf.train.ClusterSpec. A simple example of a cluster setup, one PS server with two Worker nodes, is given below:

```
cluster = tf.train.ClusterSpec({
"worker":["10.128.0.16:4327", "10.128.0.14:7279"]
"ps":["10.128.0.13:2861"]
}
```

Cluster is described as a dictionary. The dictionary contains the host machine IP address for each task. The above example gives two job types, ps and Worker, for a total of three tasks.

After the creation of the cluster, it is necessary to construct servers for each task. Each server contains the host machine's information in its cluster. Thus, servers communicate with each other within a cluster. In other words, as a server is defined, the scheme of the cluster is established. At high level, each server contains two parts: Master and Worker. Master provides the remote access by using Remote Procedure Call (RPC) protocol, which allows hosts to receive and send data packets between Workers. In addition, Master acts as a target for tf.Session. Worker is used to implement TensorFlow graph.

Tensorflow framework constructs a computational graph and uses client to interact with computing devices in the cluster. To build a distributed deep learning framework, the target argument in tf.Session needs to be specified instead of default. This server is called Master. The TensorFlow pipeline involving client and Master on both single machine and distributed system is shown in Figure 3.1.

At the beginning of the execution of TensorFlow, client is responsible for sending the computational graph and node information to master. Master takes charge of managing the computing resources. The actual computing

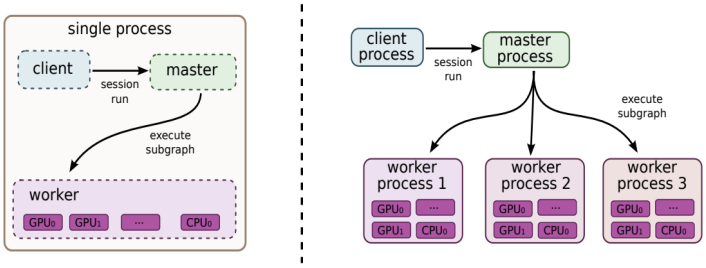

Figure 3.1: Single machine and distributed system in TensorFlow [10]

workload is processed by Workers that execute the computational tasks locally. In this way, each server includes a Master and a Worker.

At the construction of the computing graph, the cluster treats all the computing devices as local machines. Hidden from the users, the TensorFlow framework handles the actual implementation at system level. After the construction of the computing graph, we need to create a session to process the graph. Since it is a distributed architecture, we need to specify the target parameters by using the concatenation of grpc: and the host's name.

This experiment focuses on a data-parallelism distributed training strategy, which means multiple Workers apply different mini-batch data to train the same model. The gradient computed by a Worker is sent to Parameter Server to update the model globally. Thus it is necessary to replicate the model for distributed training process. TensorFlow provides some APIs to simplify the replicated training process. This experiment uses a between-graph replication mechanism to duplicate the model. Each Worker creates a client that is in the same process as the task. Each client has an identical TensorFlow computing graph. The advantage of using between-graph replication is its fault tolerance. If a Worker node fails, then the whole system is still functional and carries on the training process.

Note that, for Ring All-Reduce architecture, TensorFlow provides CollectiveAllReduceStrategy, which can be used as a parameter to pass into the tf.estimator.RunConfig.

3.5 Experiment Design

This experiment uses a mini-batch size of 32 per Worker in the training process. The size of mini-batch could affect the model's accuracy and converging speed. However, this experiment does not focus on these two factors. A mini-batch size with 32 is able to fit into a Worker's memory.

Substituting different CNN models, this experiment collects the information on performance improvements and scalability. The experimental design of substituting different models with both Parameter Server and Ring All-Reduce architectures is presented in Table 3.2. The purpose of this experiment is to investigate the performance improvements and scalability of each CNN model.

Table 3.2: Experiments design with different CNN models

Models	Batch size	Optimizer	Learning rate	Nodes
AlexNet	32	SGD	0.0001	1 to 6
VGG 16	32	SGD	0.0001	1 to 6
VGG 19	32	SGD	0.0001	1 to 6
ResNet 50	32	SGD	0.0001	1 to 6
ResNet 152	32	SGD	0.0001	1 to 6

The number of Worker nodes is one of the key parameters in this experiment. By setting different numbers of Worker nodes in the cluster, we can benchmark the communication overheads and computational power utilization of each distributed deep learning architecture. A full experimental design for distributed deep learning architecture, number of Worker nodes and batch size is presented in Table 3.3. The GCP provides node utiliza-

Table 3.3: Experiments with different Worker nodes and distributed deep learning architecture

Worker nodes	Batch size	Architecture
2	32	PS
4	32	PS
6	32	PS
2	32	Ring All-Reduce
4	32	Ring All-Reduce
6	32	Ring All-Reduce

tion and a network packet monitoring system. This experiment leverages this monitoring system to report the utilization and network packet loads for each node.

CHAPTER 4

RESULTS

This chapter presents results of using two different distributed deep learning architectures. The results include the performance improvement of each CNN model, system utilization and communication overhead. The results are shown in line charts and histograms.

4.1 Parameter Server Results

Section 2.6 introduced a widely used distributed deep learning architecture, Parameter Server. There are two types of node in the cluster of PS architecture: Parameter Server and Worker. Parameter Server is responsible for storing the parameters of the deep learning model. Worker node is responsible for calculating the gradient for the loss function of the model. In this section, we report the scalability and performance improvement of each CNN model (AlexNet, VGG16, VGG19, Resnet50, Resnet152) on a 6-node cluster deployed on GCP.

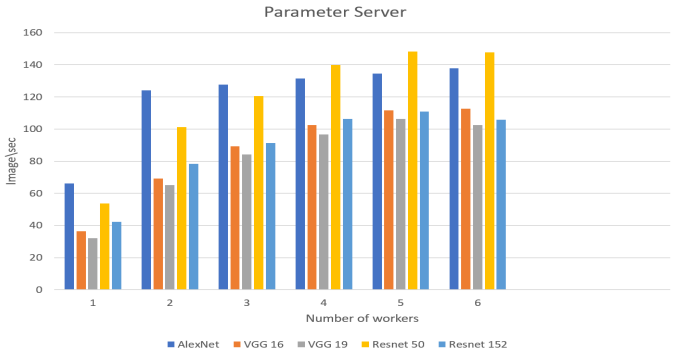

Figure 4.1: The performance of Parameter Server architecture

In Figure 4.1, to indicate the run-time performance of the models, this study uses images per second corresponding to the number of Worker nodes.

4.2 Ring ALL-Reduce Results

Section 2.7 introduced a bandwidth-optimal distributed deep learning architecture, Ring All-Reduce. Computing devices in Ring All-Reduce architecture form to a topological ring. Each device accepts the parameter information from its predecessor and sends the latest computed parameters to the successor in the ring. In this section, we report the scalability and performance improvement of each CNN model (AlexNet, VGG16, VGG19, Resnet50, Resnet152) on a 6-node cluster deployed on GCP.

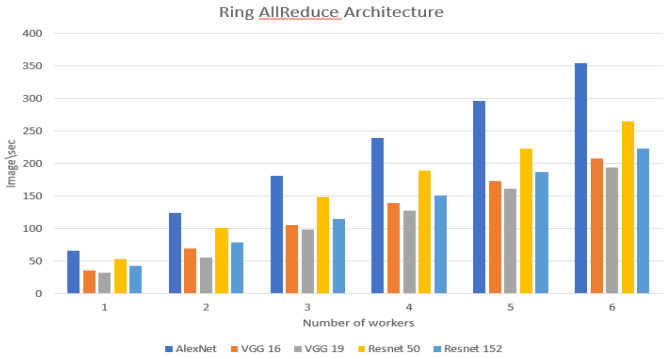

Figure 4.2: The performance of Ring All-Reduce architecture

In Figure 4.2, to indicate the run-time performance of the models, this study uses images per second corresponding to the number of Worker nodes.

To better illustrate the performance improvement of each CNN model, Figures 4.3 and 4.4 show the models speeding up information corresponding to the increment of Worker nodes. The x axes for both figures indicate how many times the architecture speeds up. The y axes for both figures indicate how many Worker nodes are in the cluster.

4.3 Utilization and Communication Overheads

By leveraging the system monitor tool, the results are gathered to show the utilization of computational resources and the number of packets sent or received by each node. Figures 4.5, 4.6 and 4.7 show the result of a 2-node setup. Figures 4.8, 4.9 and 4.10 show the result of a 4-node setup. Figures 4.11, 4.12 and 4.13 show the result of a 6-node setup. Master, Parameter Server, and multiple Worker nodes are labeled with different colors. The

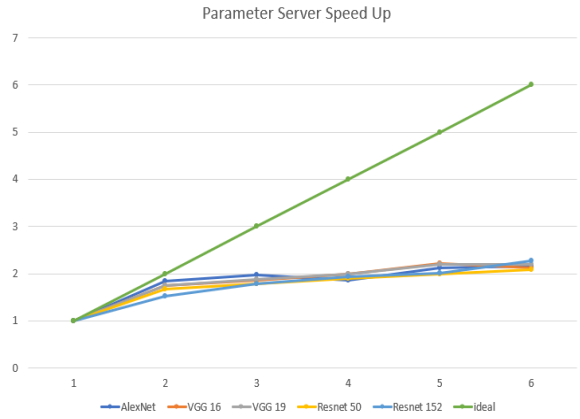

Figure 4.3: Parameter Server architecture speed-up

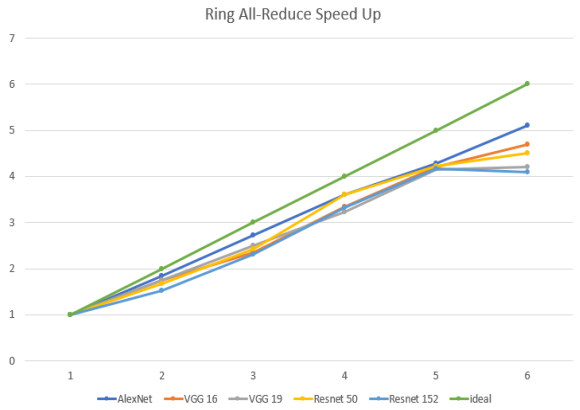

Figure 4.4: Ring All-Reduce architecture speed-up

monitored data are time series data. The y-axes in Figures 4.5, 4.8, and 4.11 indicate the fraction of computing resource utilization. The y-axes in Figures 4.6, 4.7, 4.9, 4.10, 4.12, and 4.13 indicate the number of packets received and sent among Worker nodes. Similarly, for Ring All-Reduce architecture, Figures 4.14 to Figure 4.22 show the computing resource utilization and network packets sent and received between Worker nodes.

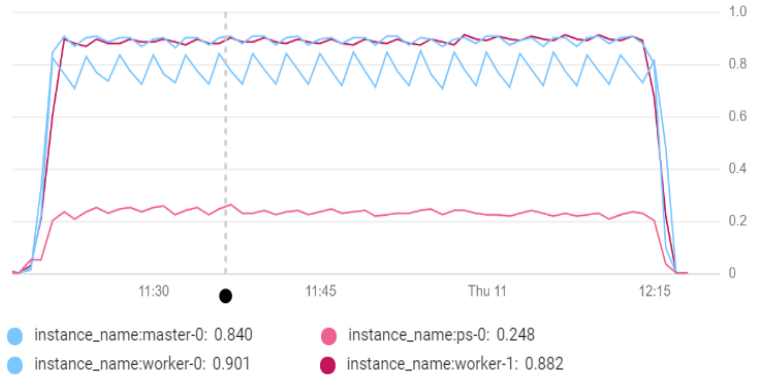

Figure 4.5: Two Worker nodes computing resource utilization in Parameter Server

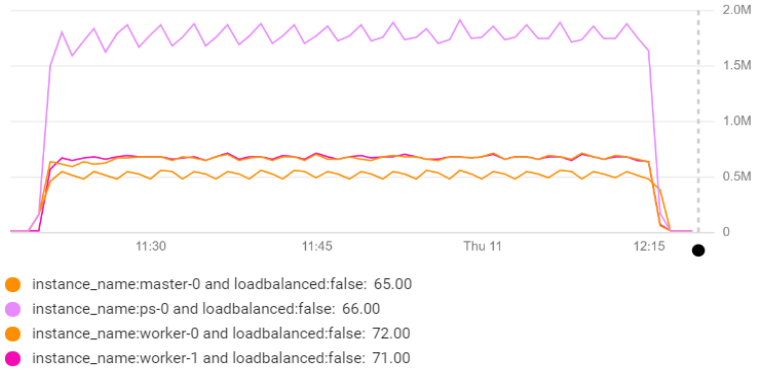

Figure 4.6: Packets received between 2 Worker nodes in Parameter Server

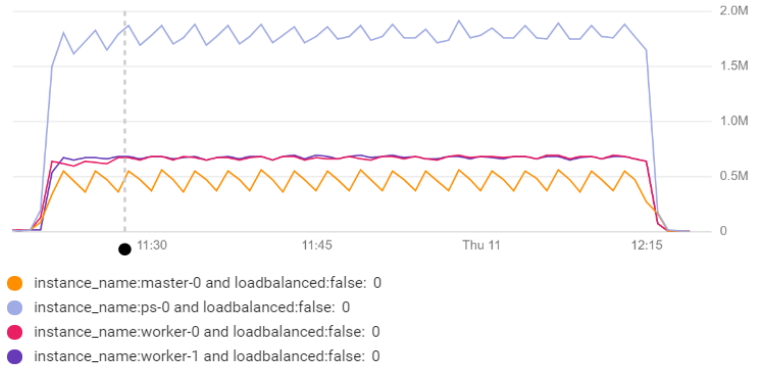

Figure 4.7: Packets sent between 2 Worker nodes in Parameter Server

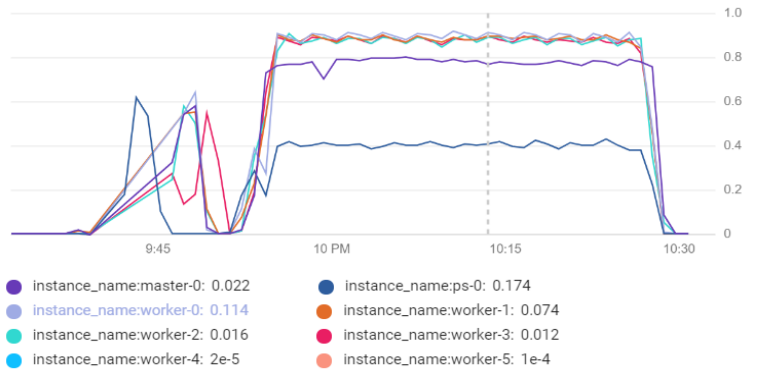

Figure 4.8: Four Worker nodes computing resource utilization in Parameter Server

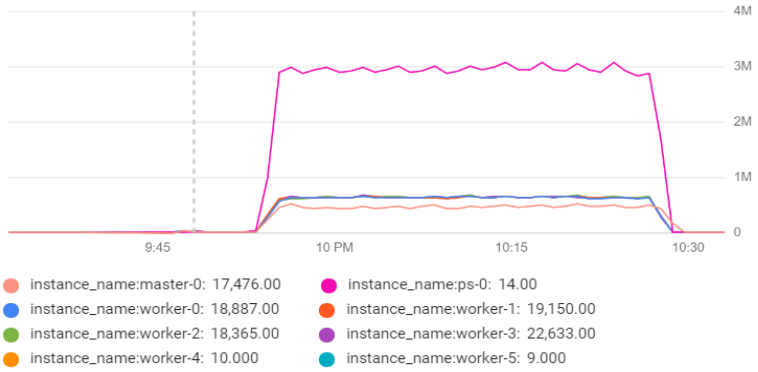

Figure 4.9: Packets received among 4 Worker nodes in Parameter Server

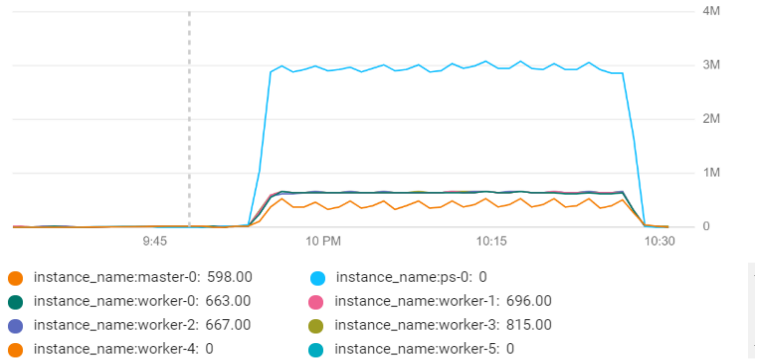

Figure 4.10: Packets sent among 4 Worker nodes in Parameter Server

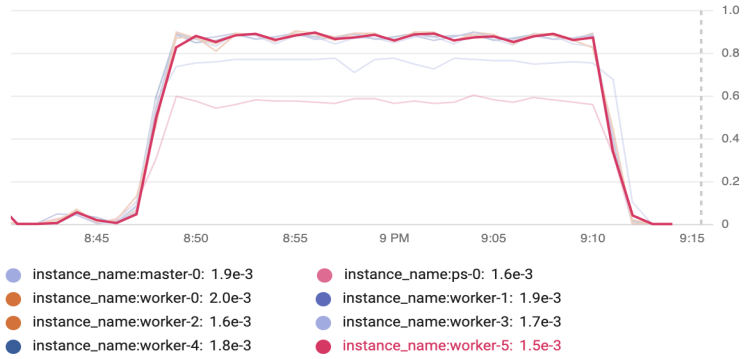

Figure 4.11: Six Worker nodes computing resource utilization in Parameter Server

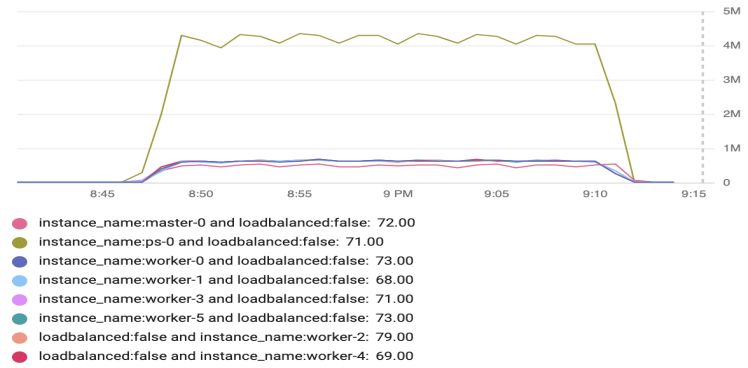

Figure 4.12: Packets received among 6 Worker nodes in Parameter Server

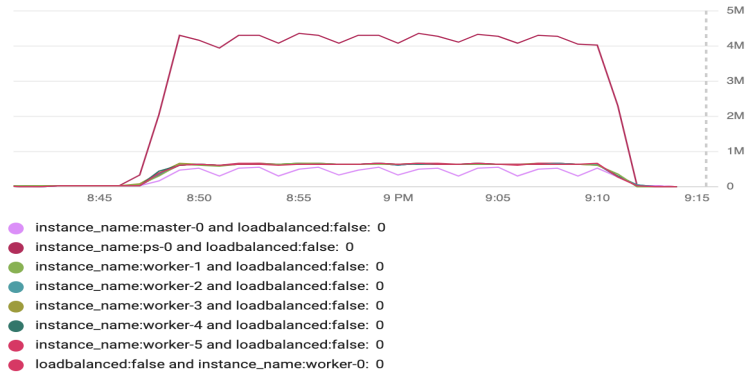

Figure 4.13: Packets sent among 6 Worker nodes in Parameter Server

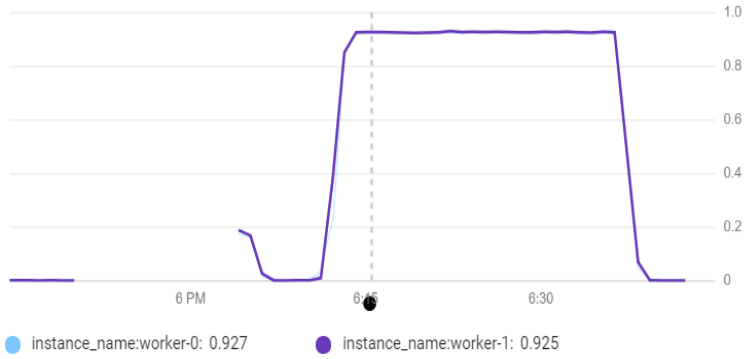

Figure 4.14: Two Worker nodes computing resource utilization in Ring All-Reduce

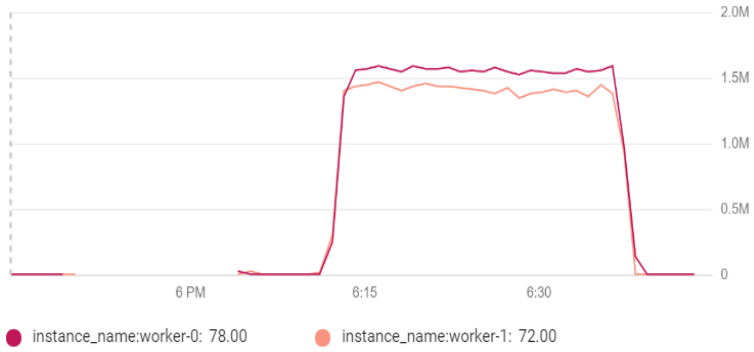

Figure 4.15: Packets received between 2 Worker nodes in Ring All-Reduce

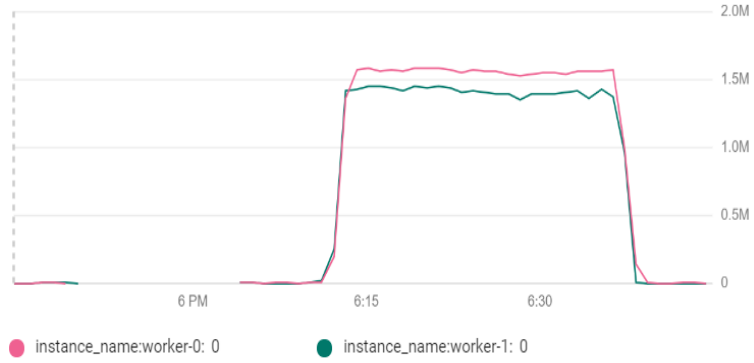

Figure 4.16: Packets sent between 2 Worker nodes in Ring All-Reduce

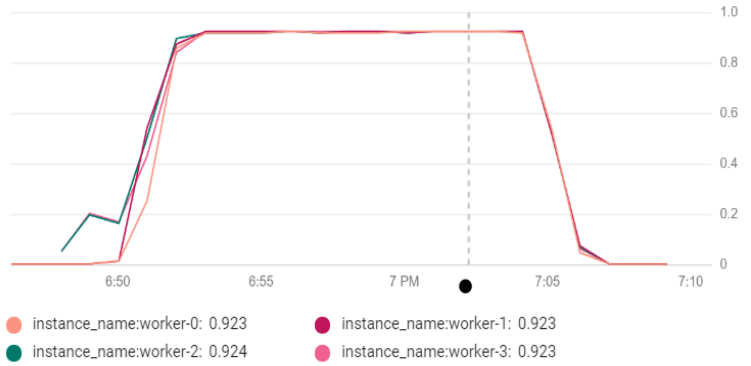

Figure 4.17: Four Worker nodes computing resource utilization in Ring All-Reduce

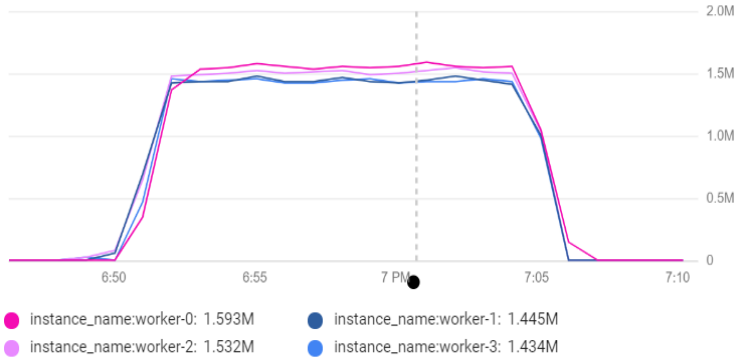

Figure 4.18: Packets received among 4 Worker nodes in Ring All-Reduce

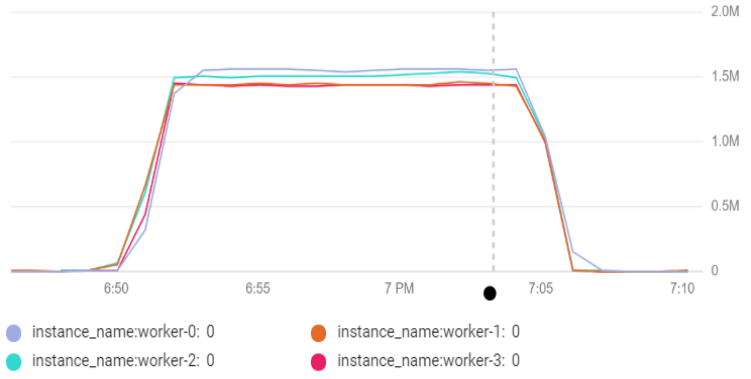

Figure 4.19: Packets sent among 4 Worker nodes in Ring All-Reduce

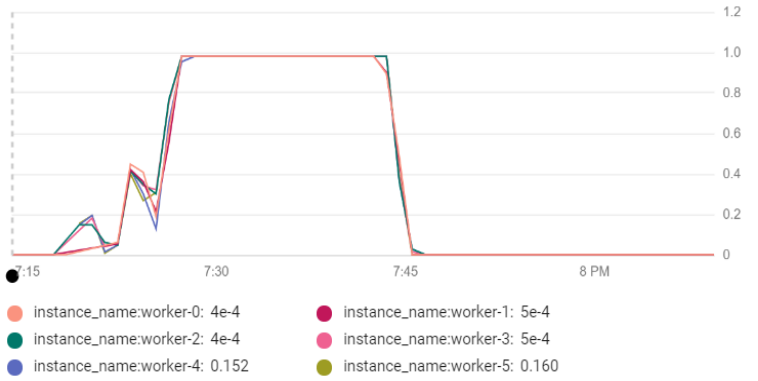

Figure 4.20: Six Worker nodes computing resource utilization in Ring All-Reduce

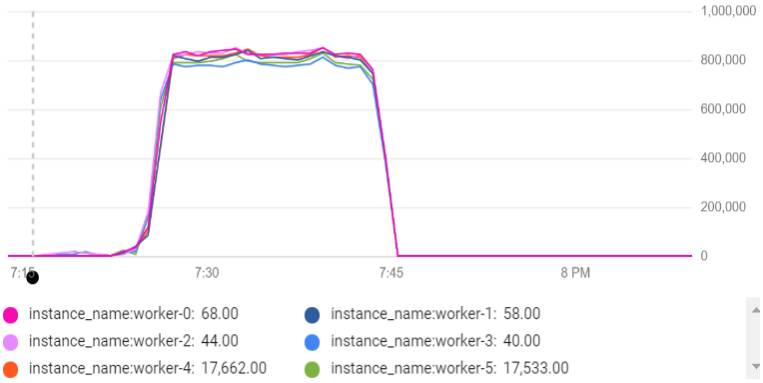

Figure 4.21: Packets received among 6 Worker nodes in Ring All-Reduce

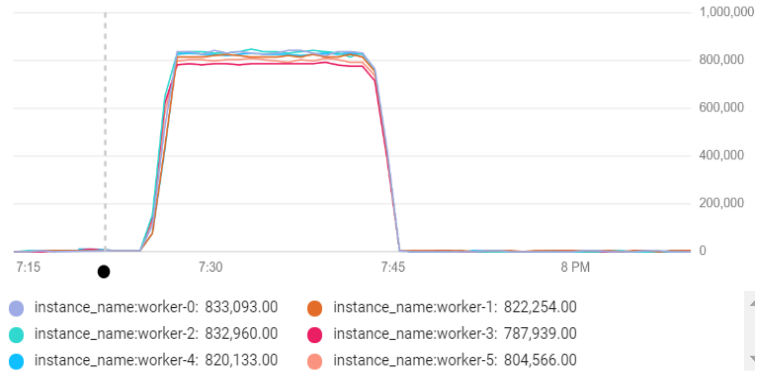

Figure 4.22: Packets sent among 6 Worker nodes in Ring All-Reduce

CHAPTER 5

CONCLUSION

Chapter 4 reports the performance improvement and scalability of multiple CNN models by applying different distributed deep learning architectures and strategies. The study also shows the computing resource utilization and communication overheads between Parameter Server (PS) and Ring All-Reduce architectures. We observed that performance improvement of training process on Parameter Server is limited. In Figure 4.3, as we increase the number of Worker nodes in the cluster under Parameter Server architecture, the speed-up of training process tends to level off. In Figure 4.4, on the other hand, the speed-up of the training process is roughly proportional to the number of nodes. This observation implies the Ring All-Reduce architecture is better at scaling out than that of Parameter Server.

The experiments are deployed on public cloud. The computing resources and network infrastructure might be variously related to assigned accessed regions and bandwidths. These factors potentially affect the accuracy of the utilization and communication overheads benchmarks. One of the approaches to address this issue is to deploy the experiment on a private cloud with more robust network access and steady computing resources.

By concatenating Figures 4.7, 4.9, and 4.11 on the same timeline, we observe that the network traffic between Worker nodes and Parameter Server node is proportional the number of Worker nodes in the cluster. In Figure 5.1, the communication overheads dramatically increase as Worker nodes are added into the cluster. However, in Figures 4.15 and 4.18, we observe that the communication overhead between Worker nodes in Ring All-Reduce architecture is roughly the same. Each Worker node fully leverages the network bandwidth. And the packets sent and received remain constant as we increase or decrease the Worker nodes in the Ring All-Reduce cluster. The packets sent and received by each Worker node only depend on the computational capacity of each Worker node.

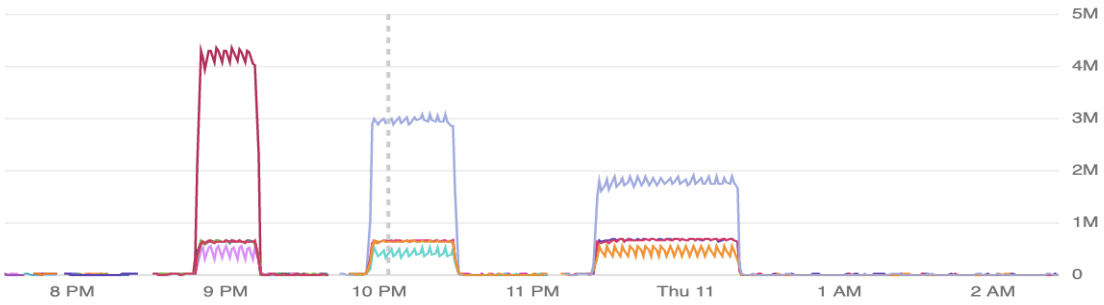

Figure 5.1: Concatenating the three Parameter Server setups (2, 4, 6 Worker nodes) on the same timeline

Another concern of this experiment is that the dataset and deep learning models we used are limited. We only used the CIFAR-10 dataset which is relatively small. A larger dataset like IMAGNET can be applied in further experiments. The CNN models are small enough that the replicas of the model can fit into a single Worker node. In future work, larger model size that exceeds the storage of a single Worker node (model parallelism) is going to be tested and benchmarked.

Lastly, some work on system fault tolerance is also worth investigating. If we break several Worker nodes during the training process in a PS architecture, what is the outcome of the trained model? What mechanisms should be implemented to overcome the offline nodes during the training process? These questions are interesting and worthwhile to answer.

REFERENCES

[1] F.-F. Li, A. Karpathy, and J. Johnson, "Cs231n: Convolutional neural networks for visual recognition 2016." [Online]. Available: http://cs231n.stanford.edu/

[2] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *CoRR*, vol. abs/1611.03530, 2016. [Online]. Available: http://arxiv.org/abs/1611.03530

[3] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[4] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, pp. 223–311, 2018.

[5] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.

[6] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2623330.2623612 pp. 661–670.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[8] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[9] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999271 pp. 1223–1231.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[11] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685095 pp. 583–598.

[12] K. Yu, "Large-scale deep learning at baidu," in *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, ser. CIKM '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2505515.2514699 pp. 2211–2212.

[13] P. Goyal, P. Dollr, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, Large Minibatch SGD: Training imagenet in 1 hour," arXiv:1706.02677, 2017.

[14] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," *CoRR*, vol. abs/1802.05799, 2018. [Online]. Available: http://arxiv.org/abs/1802.05799

[15] "Nvidia Tesla K80:the world's most popular GPU." [Online]. Available: https://www.nvidia.com/en-gb/data-center/tesla-k80/

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[19] "Intel Xeon processor e5-4620 v4." [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/xeon/e5-processors/e5-4620-v4.html